(4)

Department of Computer Science

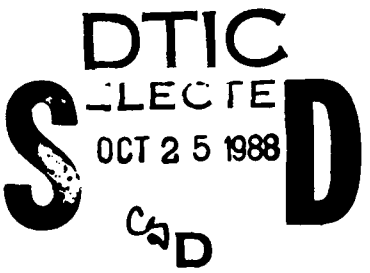
Technical Report

Propagation of Data Dependency through Distributed Cooperating Processes

Kevin L. Spier

Project Advisor
Professor Boleslaw Szymanski

Submitted to Information System Program
Office of Naval Research
Under Contract N00014-86-K-0442



Rensselaer Polytechnic Institute
Troy, New York 12180-3590

Report No. 88-24 September, 1988

Propagation of Data Dependency through Distributed Cooperating Processes

By

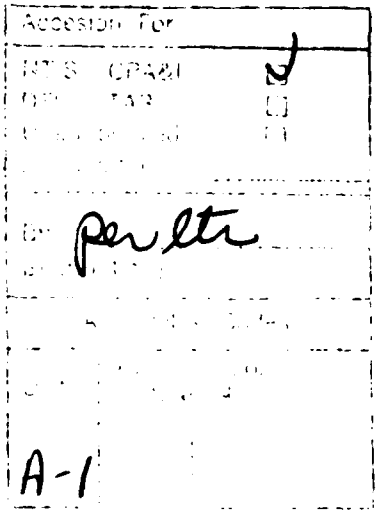
Kevin L. Spier

Project Advisor
Professor Boleslaw Szymanski

Technical Report

Department of Computer Science
Rensselaer Polytechnic Institute

Submitted to Information System Program
Office of Naval Research
Under Contract N00014-86-K-0442



ABSTRACT

EPL is an equational programming language based on MODEL. In EPL, a computation to be performed by distributed cooperating processes is described as a set of specifications. The interconnections between processes are represented as a network graph. Processes' data dependencies are derived from these interconnections. If cycles exist in the network graph (which is the usual case) these dependencies would impose additional internal scheduling constraints, created from a transitive closure of all the internal dependencies in the processes in the cycle. Therefore there is a need to derive these additional internal scheduling constraints from the internal data dependencies of all the processes and the network interconnections between them. This task requires careful analysis, which in previous versions of EPL had to be performed by the user. This thesis describes an algorithm, and its implementation, to automatically derive the additional intra-process scheduling constraints to generate the correct implementation for the target machine. This algorithm can also be used in automatic repartitioning of a process specification.

Table of Contents

Introduction

Real-time system programming is distinct from programming other parallel or distributed applications in that timing constraints are imposed on delays caused by real-time programs. Conventionally, the programmer must take into account the computer operations that cause delays, and then synchronize multiple parallel streams of data and instructions. The complexity and diversity of the skills required for this task have caused extended development times, difficulties in attaining reliable systems, and troubles in even attempting to undertake maintenance and updating of real-time systems [9].

Traditionally, every time the programmer repartitions and reorganizes his code, he must manually reanalyze his program to ensure that both functionality and timing constraints are maintained. When the user must program the control structure of processes at a low-level to handle the implementation details of communication and timing, manual repartitioning and reorganization of code become highly involved. Furthermore, the original algorithm becomes obscured by implementation details, making maintenance of the system difficult. Clearly a higher level approach is called for, in which implementation details such as synchronization and communication would be automatically generated.

Recently, several specification languages have been proposed to provide a better

bridge between the requirements of real-time systems and their respective programming [3,4,5,6]. Many of these languages support the assertive programming paradigm, in which computations are specified as sets of assertions about properties of the solution, and not as a sequence of procedural steps. In this paradigm, solution procedures are automatically generated from the assertive description of the computation. Thus, users are not involved in the implementation, whose efficiency and correctness are assured by the underlying language translator.

Depending on the type of assertions used as a basis for their notation, different languages for assertive programming have been proposed. Perhaps the best known is Prolog, in which assertions are expressed as Horn clauses. Assertive programming for parallel and distributed processing is supported by equational languages in which assertions are expressed as algebraic equations. Such languages have been proven to be an effective tool for describing general computational tasks. Programs written in equational languages are concise, free from implementation details, and easily amenable to verification and parallel processing [6]. Since the user specifies rather than prescribes the computation, equational (as well as other assertive) programs shall be referred to as specifications.

The equational language that will be discussed herein is EPL†. In EPL,

† EPL is based on the MODEL equational language [6] and is being developed at Rensselaer Polytechnic Institute

computations are specified with equations. Computations are partitioned into processes which are candidates for concurrent computation. Omitted from the computation specification are implementation details such as sequences of programming events, synchronization, and relative timing. The translation from the specification into a computation to be performed by a target computer system is performed by the language compiler.

In the EPL approach, there are two types of data dependency in a computation performed by distributed cooperating processes. One type is internal data dependency within each cooperating process. This type is identified by the EPL equational language compiler through analysis of the equational specification of a process. They are represented in the form of the *array graph*.

The other type of dependency, herein referred to as *external*, is derived from the interconnections specified in the network graph given as input to the *configurator*. If there are cycles in the network graph (which is the usual case), the external dependencies would impose additional internal scheduling constraints, being a transitive closure of all of the internal dependencies in the processes in the cycle. Therefore there is a need for automatic derivation of those additional internal scheduling constraints based on the array graphs of all the processes and the network interconnections between them.

This problem is important in automatic repartitioning and intra-process scheduling of the computation. When repartitioning a computation expressed in an equational specification, changing the scope of each process is simple. However, it is also necessary to discover externally imposed data dependencies in order to produce a correct schedule for each of the processes involved in the computation, thereby eliminating deadlock. Therefore, when topologically sorting the nodes of a process's array graph, the resulting schedule must satisfy *all* data dependency constraints.

This requires careful analysis, which in previous versions of EPL had to be performed by the user. The user must derive the dependencies by hand and then explicitly add them to the process specification with the *DEPENDS_ON* pseudo function. The goal of the implementation described in this report is to automatically derive additional intraprocess data dependency from interprocess connections in the configurator. These additional data dependencies will enable the EPL system to generate appropriate procedural programs for the target machine. Clearly, this makes the repartitioning of a computation easier and opens the possibility of building an automatic repartitioner.

The EPL Approach

The automatic implementation of a specified computation is performed by the EPL system on two levels. On the local level, the compiler accepts as input an individual process specification written by a user in the equational language. The compiler

performs completeness and consistency checks of a specification and generates a corresponding optimized sequential program in a high level language. For these purposes, the compiler builds a compact data dependency representation called an array graph.

The array graph is a concise representation of the data dependencies in the process's specification. A node in the array graph represents an entire array of data or equations, and an edge an entire array of dependencies. The underlying graph of the individual structure's elements, and their dependencies, may be derived from the array graph based on the attributes of dimensionality, ranges and forms of subscript expressions. These attributes are given for each node and edge in the array graph [8].

Typically arrays and data structures are two different ways to logically group and organize data. In EPL, a data aggregate is used to specify data structures. There are three such aggregates in EPL: *the file, the record and the group*. A hierarchy of such aggregates defines a structure. At the bottom of the hierarchy there should be arrays which are the elementary data type in EPL. Arrays consist of elements with simple types, such as integers, reals, strings, booleans, etc. Only arrays can be defined by the equations in the EPL specification. The only other elementary data type referred to in EPL equations is a subscript. In definitional interpretation of an EPL specification, subscripts are universal quantifiers. In operational interpretation, they denote array

dimensions' ranges. Each range is a finite interval of the natural numbers. There are also subscripts defined in terms of others, which are called sublinear subscripts [8]. The sublinear subscripts are convenient in defining sparse matrix operations. Subscripts imply equality of dimensions of arrays which are indexed by them in equations. Hierarchical and subscript information is interrelated, but can also occur independently of each other.

A directed edge in the array graph represents all the instances of elemental dependencies among the data elements of the nodes connected by that edge. The dependencies show precedence relations imposed on the execution order of the respective implied procedural subprocesses. Each edge in the array graph has information about the subscripts appearing with the edge's data item. This information is stored as dimension attributes, one for each subscript specified in the expression, which categorize the type of subscript expressions used.

The dimension attributes for each edge are taken directly from the expression from which the edge was derived. For each data item appearing on the right-hand side *(rhs)* of the equation (including subscript data items), there is an edge making the equation node dependent on the data item. Dimension attribute lists are formed from the subscripts used with the *rhs* data items, and stored on the data item's edge. Additionally, for every equation, the defined data item will be dependent on the equation

node and any data items used to subscript it. The edges for these dependencies will contain the subscript attributes derived from the left-hand side *(lhs)* of the equation. The attribute lists contain information categorizing the subscript expressions into one of the following types:

0 a constant was used.

1 simple subscripts; (data item I of type subscript was used).

2 simple subscript with '-1' relationship; (I-1 with I of type subscript).

3 simple subscript with '-k' relationship; (I-k with I of type subscript and k a constant, k>1).

4 simple subscript with '+k' relationship; (I+k with I of type subscript and k a constant, k>0).

5 sublinear subscript; (a sublinear subscript was used).

6 sublinear subscript with '-1' relationship; (H-1 with H of type sublinear subscript).

7 sublinear subscript with '-k' relationship; (H-k with H of type sublinear subscript, and k a constant, k>1).

8 sublinear subscript with '+k' relationship; (H+k with H of type sublinear subscript, and k a constant, k>0).

9 unknown; none of the above relationships apply.

On the global level, the *configurator* accepts as input a computation as a directed network graph, where processes are represented as nodes and their communication interconnections are shown as edges. The configurator is a compiler for the overall computation specification, written by the user in CSL (the Configuration Specification Language). The configurator must validate the CSL specification, verify

communications interfaces, synthesize the components of the network graph into an integrated system to perform the computation, optimize the concurrency of processes, and generate the procedural code necessary to implement the interprocess communications.

In EPL there are three types of files which can be used for communicating data between processes. The specification of an interface between two communicating processes is thus the structure of the interfacing file. The three file types are: sequential, direct and port. Each file has its own semantics which impose constraints on the configurator in synthesizing the computation. [6]

Sequential

The sequential file is communicated as a single entity. It implies that the file can be consumed only after it has been entirely produced. Such a file may have only one producer process, but many consumers.

Direct

In a direct file each record has a key field which is used to define (access) the record in the file. There are no restrictions on the number of producer or consumer processes and no dependencies imposed among the related processes. If only a single record is updated in a process at a time, then the EPL compiler incorporates code in the generated program to lock out other processes when updating the critical data. However, if several records must be updated together by one process, then the user is warned by the compiler and the queuing of update requests will be generated.

Port

Port files are used to provide a concurrent interface between processes. There are no restrictions on the number of producers or consumers of a port file. An entire record is communicated at a time.

User drawn edges in the network graph connecting two processes' port files imply the propagation data dependencies. When these additional data dependencies affect the behavior of a process in the network, it is necessary to impose additional constraints on the possible procedural implementation for the process to prevent deadlock in the computation.

Need for External Data Dependencies

Let's consider the generation of procedural code from an EPL specification. In the case where there is no internal data dependency of implied read on implied write, we can have two seemingly independent statements, where one reads from an input port file and the other writes to an output port file. Due to the lack of dependencies, the semantics of EPL allow the compiler to decide the order in which to schedule the execution of these statements in a process. It is possible for the read to become scheduled before the write. Now, let's assume that some external dependency exists, as expressed in the configuration, causing the read to depend on the write. This would force the process and subsequently the entire system into a state of deadlock.

One might suggest that if the compiler always scheduled all port file writes before all port file reads (except when an internal dependency of write on read exists) a correct schedule would always be produced for each process. However, such a solution is unacceptable for the following reasons.

Requesting that all port file writes must occur before reads, unless an internal dependency exists, is equivalent to drawing an external dependency of read on write, no matter whether it really exists. Since scheduling and optimization must preserve all dependencies in the array graph, adding additional dependencies limits the space of feasible solutions. Therefore accepting the suggested solution may limit the scheduler and optimizer and thus lead to generating programs less efficient then without this limitation.

As an example let's consider a specification as in Figure 1. If the scheduler is free to order operations on files F1 and F2 independently of each other, then the program consists of two parallel loops (see Figure 2). No synchronization is involved. However, if we force all writes before reads then the program consists of one loop in which evaluation of function *func* for two arguments is done in parallel (see Figure 3). These parallel evaluations are synchronized in each loop step. To see that the second solution is less effective, let's assume that function *func* is evaluated in one unit of time for odd arguments and three units of time for even arguments. Let files F1 and F2

Fig. 1 Program A (EPL specification)

```
Process: M;
input F1, F2;
output F1, F2;

file: F1 (port),
  2 record r1[*],
    3 int i1;

file: F2 (port),
  2 record: r2[*],
    3 int: i2;

subs: i;

if (i == 1) then out.i1[i]=1; else out.i1[i]=func(in.i1[i-1]); end if;
if (i == 1) then out.i2[i]=1; else out.i2[i]=func(in.i2[i-1]); end if;
```

contain 2n numbers, n even and n odd, placed in such an order that even numbers in F1 are opposite odd numbers in F2. If the function *func*'s evaluation dominates the execution time, then the first implementation will take 4n units of time and the second one 6n units of time.

Now let us examine the case where the external dependency exists between two communicating processes. The configurator would produce code that will deadlock, instead of producing an error message (see Figure 4). Thus, the necessity for inferring

Fig. 2 Program A (implementation without dependency

```
fork:
  begin
    i := 0;
    while ( !eof(F1) ) loop
      i := i+1;
      if (i == 1) then i1:=1 else read(i1); end if;
      write(func(i1));
    end loop
  end
  begin
    i := 0;
    while ( !eof(F2) ) loop
      i := i+1;
      if (i == 1) then i2:=1 else read(i2); end if;
      write(func(i2));
    end loop
  end
:join
```

external data dependencies is evident.

The External Data Dependency Analyzer (EDDA)

The EDDA derives external data dependencies by performing two levels of analysis. At the process (local) level, the internal data dependencies between input and output port files (IDDBIOPS) are determined. These local port file dependencies specify paths through which external data dependencies may be propagated based on

Fig. 3 Program A (implementation with dependency)

```
i1 := i2 := 1;
while ( !eof(F1) && !eof(F2) ) loop
  fork:
    write(func(i1));  write(func(i2));
  :join
  fork:
    read(i1); read(i2);
  :join
end loop
```

the given configuration specification. At the configuration (global) level, the results of the individual process analyses are combined with the configuration specification to compute the external data dependencies for each of the processes of the computation.

Since external data dependencies are derived only after the local analysis of each process participating in a computation has been performed, the overall operation of the EPL system must change. Specifically, with the introduction of the EDDA comes a new relationship between the compiler and configurator. A proposal for the reorganization of the EPL system follows.

Fig. 4 Program B

```
/* Configuration specification */
M1.F1 -> M2.F2 -> M1.F1;

/* Specification */
Process: M1;                    Process: M2;
  input: F1;                     input: F2;
  output: F1;                     output: F2;

file: F1 (port),                file: F2 (port),
  2 record: r1[*],                2 record: r2[*],
    3 int: i1;                      3 int: i2;

out.i1 = in.i1;                 out.i2 = in.i2;

/* Implementation */
while ( !eof(F1) ) {            while ( !eof(F2) ) {
  read(i1);                       read(i2);
  write (i1);                     write(i2);
}                               }
```

Both M1 and M2 wait in read for message from the other process resulting in a classical deadlock.

The new EPL system

External data dependencies influence the intra-process scheduling of equations, and therefore must be derived before the high level language program which represents the process is generated. Furthermore, it is necessary to produce a complete, correct

array graph for each process before the IDDBIOPS can be derived. Given these requirements a division of the compiler into two parts or phases, which are referred to as Comp1 and Comp2, is necessary. Comp1 generates an array graph from process's specification. The array graph is used in analyzing the process's IDDBIOPS. Comp2 generates the actual high level language program that implements the process based on its array graph and its external data dependencies, once they have been computed (compare Figure 5).

In Comp1, the process specification is processed and then an analysis of the process's IDDBIOPS is performed. Every process must go through this phase of compilation before proceeding to phase Comp2. After every process has gone through Comp1, the external data dependencies for each process can be derived from the combination of the configuration specification and all of the individual processes' IDDBIOPS.

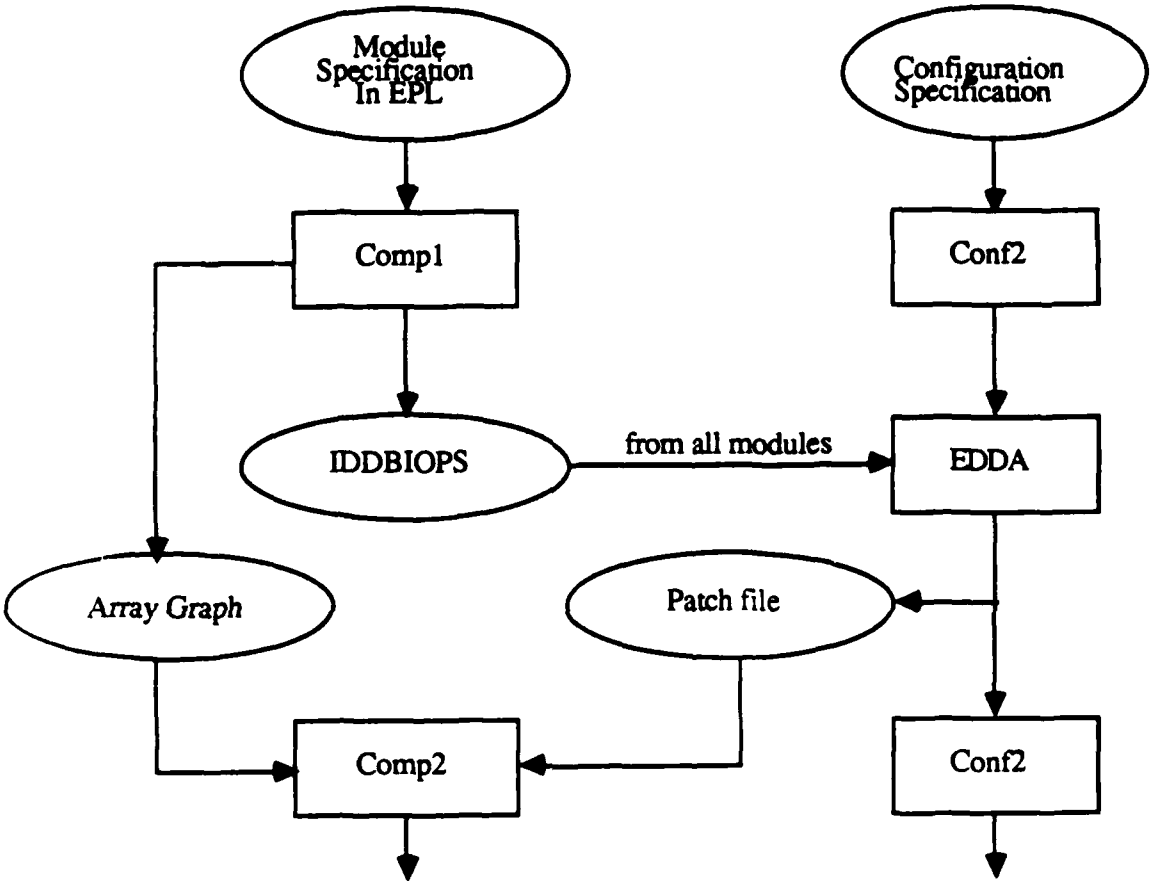
The format of the input to Comp2 is an important issue. As previously stated, the input to Comp2 is an array graph and external data dependencies for an individual process. Before equation scheduling can occur the external data dependencies must be added to the array graph as dependency edges. Having them recorded separately from internal dependencies is advantageous for the following reason. Suppose we were to allow the EDDA to directly alter a process M's array graph. Each time the

configuration specification or the specification of a process which previously caused an external data dependency to be imposed on M is changed, it would be necessary to force all the processes to go through Comp1 again. This would be necessary since we could no longer have the original array graph for each process. Therefore the external data dependencies are recorded as a *patch* to the array graph.

Given the requirements of the compiler, the automatic derivation of external data dependencies also affects the operation of the configurator. The configurator must also be divided into two parts or phases, called here Conf1 and Conf2. In Conf1 the configuration specification is translated to a network graph description usable by the EDDA and Conf2. In Conf2 the configuration specification is validated and the script is generated to setup, invoke and oversee the entire computation. Conf1 and Conf2 must be separate phases since only after Comp1 and Conf1 had run could Conf2 be able to verify all interfaces and check that all the participating processes are correct.

After every process involved in the computation (participation is derived from the configuration specification) has gone through Comp1, and their respective external data dependencies have been derived, their respective array graphs and *patch* files may be passed on to Comp2. Figure 5 gives an overview of the new EPL system, showing this interdependency between the configurator and compiler.

Fig. 5 Diagram of New EPL System



It is interesting to note that if the IDDBIOPS analysis following phase Comp1 determines that the given process cannot be affected by external data dependencies it may then continue directly onto Comp2. This is the case when the given process does not have at least one input and one output port or, alternatively, a single input-output port. Thus, a specification which contains only input, only output, or no port file

declarations may proceed directly onto Comp2.

EPL vs. Ada

Several features of EPL facilitate parallel and distributed programming, and especially real-time system development. There are, however, a number of languages intended for the design, development and maintenance of real-time and other parallel and distributed systems. The most prominent among them is perhaps Ada‡. In this section, EPL will be compared with Ada to contrast their respective paradigms and programming idioms. It should be immediately noted that it is relatively easy to generate Ada code from an EPL specification, but quite difficult to do it the other way around.

Many of the features of Ada software development can also be found in EPL. EPL supports the notion of separate compilation for processes much the same way Ada does for *library units* (packages, tasks and top-level procedures). In EPL, the specification need only go through the entire compilation process once unless it has changed, in which case the EDDA can be used to determine which processes' external data dependencies have been affected, requiring them to go through Comp2 again. In Ada, when a package specification is changed, the package and all dependent library units must be totally recompiled. Although it is not possible to compare the effect of changing a specification in EPL and in Ada, we can see that the declarative nature of

‡ Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

EPL allows a more efficient response to the change.

While both Ada and EPL support real-time programming, the Ada approach is at too low a level for many cases. When programming in Ada, the programmer has to be concerned with low-level implementation details. In contrast, the EPL approach is declarative, ie. no control structure information is supplied in the specification of a process, it is instead generated from the explicit and implied data dependencies. The Timing Constraint Analyzer [7] can be used with the scheduler and code generator to generate a procedural implementation that meets the given time constraints while keeping them out of the actual process specification.

Both Ada and EPL support *top-down* and bottom-up design of a computation performed with communicating sequential processes. However, EPL automatically performs deadlock detection, timing constraint analysis, synchronization, code generation of communications details and configuration synthesis. In Ada these must be explicitly coded and analyzed. Clearly, the EPL approach shows much promise in the area of rapid prototyping.

In EPL, the configuration and process specifications may be developed independently of each other. Thus, the user can test the configuration to see if a proposed solution is correct even before writing any specifications describing the individual processes. The opposite is also true, allowing the user to change the individual process

specifications without having to know the configuration. The user may also change the configuration specification many times during development to see how it affects the overall performance of the computation.

This level of independence cannot be achieved in Ada although both top-down and bottom-up design are possible. This is because the concept of a configuration specification in EPL is absent from Ada. Thus, configuration information must be physically encoded and distributed over the tasks in the computation described in Ada. The specification of relationships between processes participating in a computation in EPL are reflected by a straightforward network graph making debugging, reorganization and prototyping simple. In Ada, these tasks require the user to perform careful and tedious analyses. Furthermore, changing the configuration of a computation in EPL only requires the process scheduling to be recalculated for processes whose external data dependencies are affected. These processes are identified by the EDDA and then pass through Comp2. The results of such a change in a computation described in Ada are unpredictable! The Ada user must perform careful analysis to determine what manual recompilation and process editing must occur.

In EPL synchronization is achieved by the unblocking send and blocking receive. This method of synchronization can be used to implement the equivalent of the Ada *rendezvous*, and visa versa. However, the EPL method allows for a simple pipelining

schema where writers always write to the receiver's input queue rather than having the processes interchange data directly. In Ada, the programmer is allowed to specify the information to be communicated between processes (tasks), as well as any computation that can occur while the processes are locked during the rendezvous. While it might be argued that this makes the rendezvous a more powerful mechanism, it also lessens the independence of the computations described by each task.

The reason is that one of the processes involved in rendezvous can *hold up* the others by not releasing them directly after exchanging data. In EPL, we can achieve an effect similar to the rendezvous by requesting that the recipient of the message sends an acknowledgement to the sender. This will lock the sender until the recipient receives (and possibly processes) the message. On the other hand, simulating the blocking receive and unblocking send with the Ada rendezvous mechanism requires the creation of additional tasks performing the buffering between communicating processes. Clearly, the system overhead caused by (additional) acknowledgement messages is much less severe than that caused by the creation of additional tasks. In that sense, we may argue that the EPL communications primitives are more efficient than the Ada rendezvous.

The Dining Philosophers Example

The specification of data dependency in EPL differs greatly from that of conventional procedural programming. The example described here is used to present how data dependency is specified in EPL at the process (local) and configuration (global) levels and the importance of the EDDA. The example is of a resource allocator and processes requesting resources, such as found in operating and real-time systems.

Resource allocation captures the essence of many concurrent systems used in real-time applications. Many strategies exist for allocating resources. The strategy selected here avoids deadlocks by requiring that a process submit a request for all the resources that it will need, and release them when not further needed. Indefinite postponement is also prevented by preserving the order of arrival of requesting messages.

In order to make the example more specific and easier to follow, it is stated in terms of the *Dining Philosophers problem.*

> Five philosophers, who represent individual processes, share a circular dining table where each has an assigned seat. There is one fork between each two seats. A philosopher needs the forks to his right and left in order to dine. A philosopher desiring to dine requests the forks. When available, the resource allocator issues both forks and the philosopher proceeds to dine. When finished, he releases both forks, which become available to his immediate neighbors on a first-come-first-allocated basis. [5]

In the example, the five philosophers and the resource allocator form respective processes naturally. Processes are producers and/or consumers of their respective port

files. The configuration for the Dining Philosophers problem is shown in Figure 6. The consumer/producer relationship is represented by a directed edge in the network. Each philosopher process (P[I] for I = 1 to 5) produces throughout its life a file of requests and releases of resources (REQ_REL) and consumes a corresponding file of allocations of resources (ALLOC). The resource allocator (R) has an output port file of allocations (ALLOC) and an input port file of requests and releases of resources (REQ_REL). A description of the external dependency between the processes is given in figure Figure 7. The EPL specifications for the philosopher and resource allocator processes are given in Appendices I & II.

The Algorithm

As previously stated, the EDDA requires as input the IDDBIOPS for all of the processes and the configuration specification. The purpose of the IDDBIOPS algorithm is to find all the known read before write dependencies of output port files on input port files. The function Get_Process_IDDBIOPS (see Figure 8) will determine the IDDBIOPS for a given process and will have a return value which indicates whether or not the compilation of this process may proceed directly onto Comp2 (such is the case when either the number of input or output ports is zero). This is accomplished by collecting all input fields for each input and input/output port file and tracing them through the array graph to the dependent fields of output or input/output port

files. If a path exists between an input and an output port, then the output port is said to be dependent on the input port. The type of dependency will be the maxima, or worst case, of dependency computed by the transitive closure of all possible paths from the input port to the output port. Such edges can be used to propagate external data dependencies in the *super graph.*

The super graph is a concise description of the inter-process data dependencies in a computation. The nodes of the super graph are the processes participating in the computation. The super graph's edges are taken from the configuration specification, in which they are specified as the relation of an output port file to an input port file.

Although the analysis of dependencies described so far is performed at the field level what is needed are record dependencies. For sequential files this is not relevant since the entire file must be written before in can be read by another process. For port and direct files, however, messages passed between processes are records and not fields. Therefore, those dimensions of each array graph edge which relate to the parent input record of the source input field need be considered. Thus, given the input port declaration in Figure 9, when considering any edge on a path from input field Z (to some output field), we only consider those edge dimensions which *correspond to* the two record dimensions of Y. An edge dimension is said to correspond to another edge dimension if and only if both dimensions belong to the same range set [2].

Fig. 6 Configuration for Dining Philosophers Example

```
/* Configuration */
RANGE.P = 5;

R.ALLOC -> P.ALLOC;
P.REQ_REL -> R.REQ_REL;
```

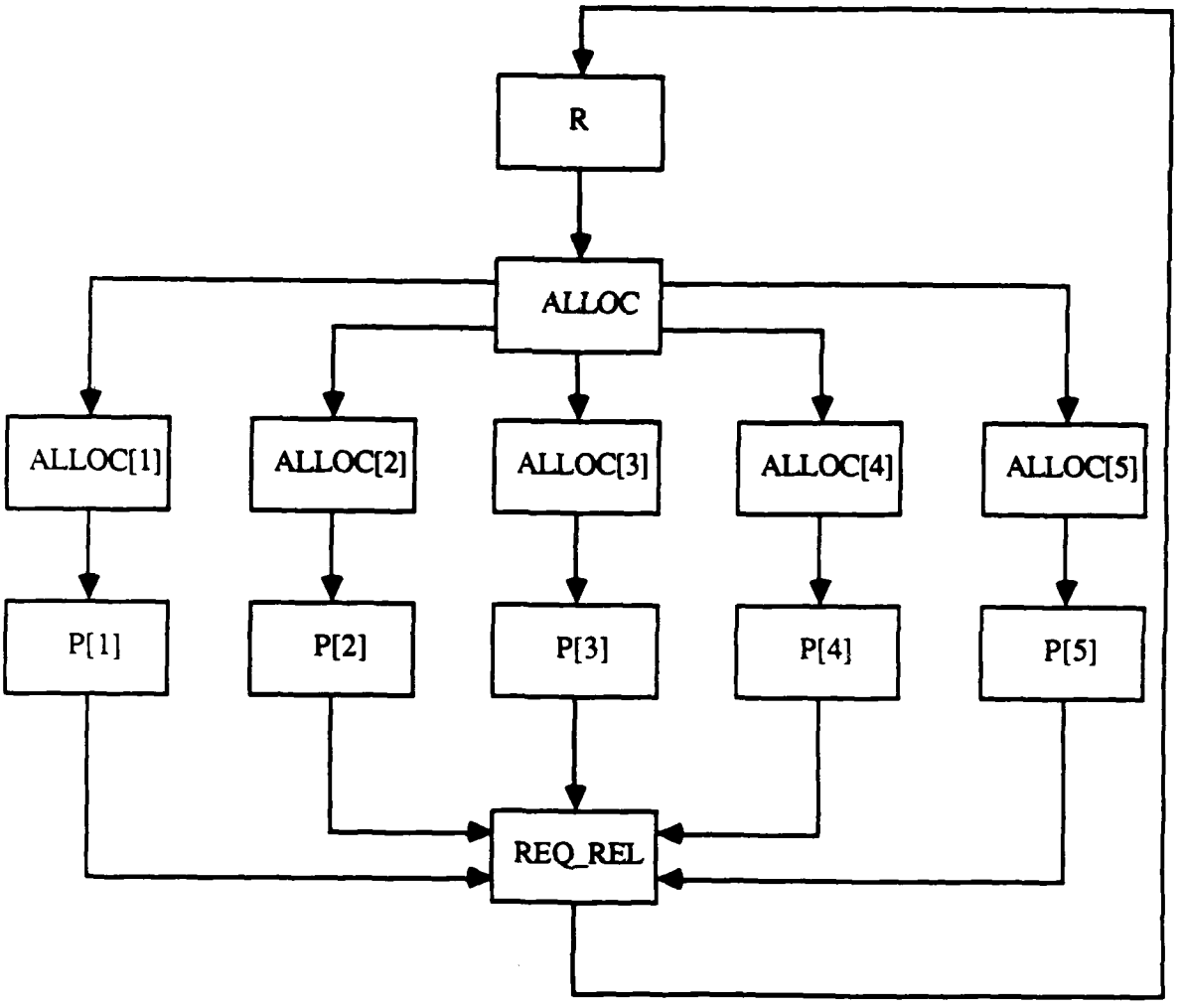
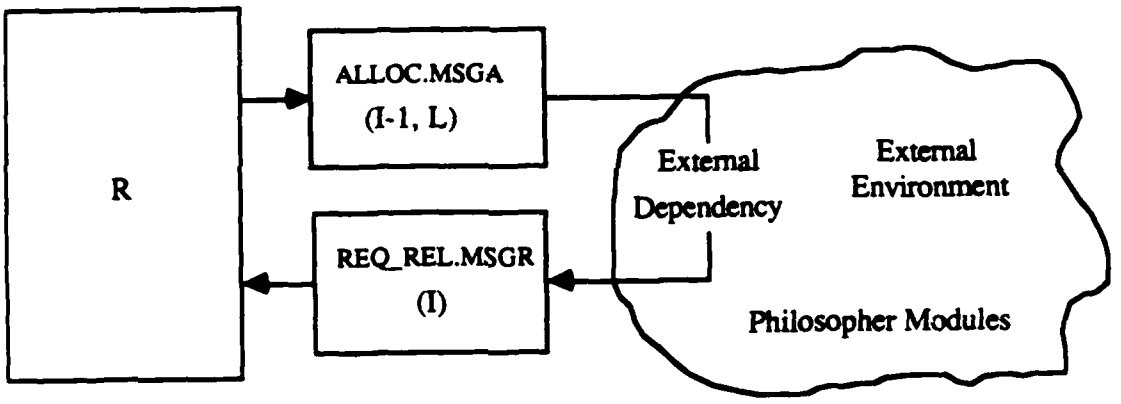


Fig. 7 Diagram of external data dependency in Philosophers



The algorithm computes the type of dependency between input and output ports by performing a data flow analysis through the set of assertions connecting the input and output ports. Aside from actual assertions the only other EPL entities which must be considered in this analysis are LAST and RANGE variables. The RANGE variables which are defined by equations do not require special treatment since their defining equations are already included in the data flow analysis. Other RANGE variables have to be defined through a declaration or End Of File (EOF) condition. In the former case they are just constants, and in the latter case they have to be scalars, therefore in both cases they may not be considered at all. Only LAST variables has to be treated as

Fig. 8 Function Get_Process_IDDBIOPS

```
Function Get_Process_IDDBIOPS( process )
  input_port_count := output_port_count := 0;
  FOR out_port_file IN Output_Port_Files( process ) LOOP
    output_port_count := output_port_count + 1;
    Output 'O', File_Name( out_port_file ), Dimensionality( out_port_file )
  END LOOP
  FOR in_port_file IN Input_Port_Files( process ) LOOP
    input_port_count := input_port_count + 1;
    Output 'I', File_Name( in_port_file ), Dimensionality( in_port_file )
    field_list := NULL
    FOR record IN Input_Records( in_port_file ) LOOP
      COLLECT Get_Fields( record ) INTO field_list
    END LOOP
    dependent_output_port_files_list := NULL
    FOR field IN field_list LOOP
      Trace_Thru_Array_Graph( field );
    END LOOP
    FOR out_port_file IN dependent_output_port_files_list LOOP
      Output 'O', File_Name( out_port_file ), How_Dependent( out_port_file )
    END LOOP
  END LOOP
  RETURN (input_port_count != 0 && output_port_count != 0);
END Get_Process_IDDBIOPS;
```

special cases. Note that since the analysis is starting and ending at the field level of port files, hierarchical dependencies need not be considered.

In performing the data flow analysis through the set of equations which connect

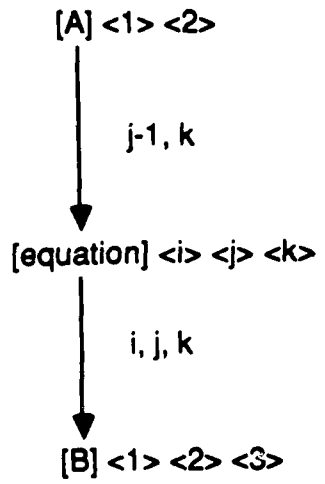
Fig. 9 Sample Input Port File Declaration

```
file: IN_FILE (port),
  10 group: X[*],
    20 record: Y[*],
      30 int: Z[10];
```

input and output ports we must determine the correspondence between dimensions of a rhs source and the lhs target. It is clear that this relationship may only be derived by symbolically matching subscript references. Only those dimensions on the rhs source which correspond to input port record dimensions need be considered.

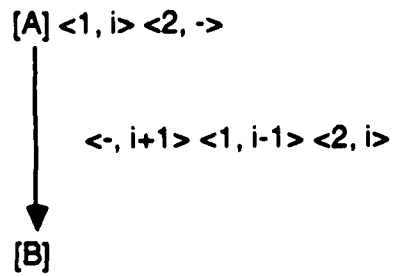
For example, an equation B[i,j,k] = ...A[j-1,k]... generates the subgraph shown in Figure 10. The first dimension of B would not be matched to any of the dimensions of A, thus an I+1 dependency exists on this dimension of B (denoted <-,i+1>. The second dimension of B would be matched to the first dimension of A, thus an I-1 dependency exists on this dimension of B (denoted by <1,i-1>). Finally, the third dimension of B would be matched to the second dimension of A, thus an I dependency exists on this dimension of B (denoted by <2,i>). This derived relationship between A and B is shown in the reduced subgraph in Figure 11.

Fig. 10 Subgraph for equation B[i,j,k] = ...A[j-1,k]...



It is important to understand three points being made here. First, this new direct edge describes the equation implied dependency between the source (A) and the target (B) and not an absolute one. Secondly, since the first dimension of B is unmatched this means that the entire range input needed for computing the first dimension A must first be read before B can be computed. Finally, the result of symbolic matching is a reduced array graph which may be uniformly searched. This is true because the nodes of this graph are the field nodes of the original array graph and its edges are labeled with the dependency between dimensions of the rhs source and the lhs target of asser-

Fig. 11 Reduced subgraph for equation B[i,j,k] = ...A[j-1,k]...



tions.

However, the symbolic correspondence may not always be so easily determined. Such is the case when references to sublinear subscripts appear or when multiple subscripts belong to the same range set. Thus, the algorithm for performing symbolic matching must be able to handle these special cases. The way in which matching is performed is determined by the type of the rhs source, ie. whether it is a scalar or an array (see Figure 12).

When the rhs source is a scalar the equation must be in one of the following two forms: (1) x = ...y... or (2) a[...] = ...y... Since a scalar may only be assigned a single value at a time, the dependency of the lhs target on the such a rhs source must be of type I. The algorithm for this case is given in Figure 13. When the rhs source is an

Fig. 12 Function Symbolic_Matcher

```
FUNCTION Symbolic_Matcher( in_edge, out_edge )
  in_ndims := Dimensionality(in_edge->srce_node);
  IF (in_ndims == 0) THEN
    RETURN Symbolic_Matcher_RHS_Scalar( in_edge, out_edge );
  ELSE
    RETURN Symbolic_Matcher_RHS_Array( in_edge, out_edge, in_ndims );
  END IF
END Symbolic_Matcher;
```

array the complexity of the algorithm increases dramatically. The algorithm for this case is given in Figure 14.

The other key components of the algorithm for data flow analysis are described in the procedure Trace_Thru_Array_Graph (see Figure 15) and the functions Type_Thru_Edge (see Figure 16), Combine_Mult_Direct, Add_Out_Dependency, and Combined_Edge_List. The procedure Trace_Thru_Array_Graph is invoked recursively to traverse the array graph and collect the dependent output ports for a given input port into the list *dependent_output_port_files*.

The function Combined_Edge_List (see Figure 17) returns the list of unique edges outgoing from the given data node to any data node defined by it (via an assertion). Multiple direct edges between these nodes are combined into single edges

Fig. 13 Function Symbolic_Matcher_RHS_Scalar

```
FUNCTION Symbolic_Matcher_RHS_Scalar( in_edge, out_edge )
 out_ndims := Dimensionality(out_edge->dest_node);

 /* case when lhs is scalar too */
 IF (!out_ndims) out_ndims:=out_ndims+1; END IF

 edge_tok = mk_dfa_token(out_ndims);
 FOR i IN 0..out_ndims-1 LOOP
  edge_tok[0].in_dim = i;
  edge_tok[0].type = RANK_I;
 END LOOP

 RETURN edge_tok;
END Symbolic_Matcher_RHS_Scalar;
```

labeled with the maximum type of each of their dimensions. Since all of this information is static a simple caching mechanism is used to avoid expensive recomputation each time the given data node is visited. Thus, three reductions on the search space for the data flow analysis are performed: (1) elimination of edges and nodes in the array graph which cannot contribute to data flow; (2) multiple direct edges between nodes are combined into single edges; (3) direct edges are drawn between source and target nodes for all assertions.

Fig. 14 Function Symbolic_Matcher_RHS_Array

```
FUNCTION Symbolic_Matcher_RHS_Array( in_edge, out_edge, in_ndims )
 in := Get_Edge_Dimension_Vector(in_edge);
 out_ndims := Dimensionality(out_edge->dest_node);
 edge_tok := Mk_DFA_Token(out_ndims);
 i := 0; last_match := -1;

 FOR p IN Get_Edge_Dimension_Vector(out_edge) LOOP
  osub := osub2 := Get_Subscript( p );
  IF (osub != NULL) THEN
   osubl := (Is_Sublinear(osub) ? osub->sibling.l : NULL);
   REPEAT
    IF (osubl != NULL) THEN osub2=osubl->st, osubl=osubl->next; END IF
    FOR j IN (last_match+1)..(in_ndims-1) LOOP
     isub := isub2 := Get_Subscript( in[j] );
     IF (isub != NULL) THEN
      isubl := (Is_Sublinear(isub) ? isub->sibling.l : NULL);
      REPEAT
       IF (isubl != NULL) THEN isub2=isubl->st, isubl=isubl->next; END IF
       IF (isub2 == osub2) THEN
        edge_tok[i].in_dim := last_match := j;
        edge_tok[i].type := New_Path_Type(Rank( in[j] ), Rank( p ));
        isubl := osubl := NULL;
       END IF
      UNTIL (isubl == NULL);
     END IF
     EXIT WHEN (last_match == j);
    END LOOP
   UNTIL (osubl == NULL);
  END IF
  i := i + 1;
 END LOOP
 RETURN edge_tok;
END Symbolic_Matcher_RHS_Array;
```

Fig. 15 Procedure Trace_Thru_Array_Graph

```
PROCEDURE Trace_Trough_Array_Graph( node )
 /* note that NODE is the last node reached in current path */
 Mark(node);

 REPEAT
  FOR p IN Combined_Edge_List(node) LOOP
   IF ((next_node := Type_Thru_Edge(p)) == NULL) THEN
    CONTINUE;
   ELSIF (next_node == node) THEN
    EXIT;
   ELSE
    Trace_Thru_Array_Graph(next_node);
   END IF
  END LOOP
 UNTIL (p == NULL || next_node != node);

 UnMark(node);
END Trace_thru_Array_Graph;
```

In a correct array graph, multiple direct edges between two nodes must have corresponding, and an equal number of, dimensions. Therefore, multiple direct edges can be represented as a single edge with corresponding, and an equal number of, dimensions, the type of each dimension being the type of the maximum dependency of each dimension over all the edges. The following order of dependency types is used: I+1 (subscript types 0,4,8,9), I (subscript type 1,5), I-1 (subscript types 2,3,6,7). The

Fig. 16 Function Type_Thru_Edge

```
FUNCTION Type_Thru_Edge( edge )

  src_node := Source( edge );
  dest_node := Target( edge );
  old_ttok := Get_DFA_Token( dest_node );
  new_ttok := Compute_New_DFA_Token(src_node, edge);

  IF (Marked_P(dest_node)) THEN
    /* a back edge exists, the question is how. if the back edge results
       in a new I+1 dependency for target we must reset trace in order
       to correctly type all paths emanating from the target */
    IF (! Edge_Contributes_New_Back_Edge_Relation( old_ttok, new_ttok)) THEN
      dest_node := NULL;
    END IF
  ELSIF (old_ttok == NULL) THEN
    Set_DFA_Token(dest_node, new_ttok);
  ELSE
    /* No back edge and 'dest_node' has been visited sometime during DFA */
    old_ttok := Combine_DFA_Tokens(old_ttok, new_ttok);
  END IF

  /* if target is an output field record its current token value */
  IF (dest != NULL && Is_Out_Field(dest_node))
    Add_Out_Dependency(dest);
  END IF

  RETURN dest_node;
END Type_Thru_Edge;
```

Fig. 17 Function Combined_Edge_List

```
FUNCTION Combined_Edge_List( node )
 IF ((cache := Get_Combined_Edge_List_Cache(node)) == NULL) THEN
  FOR q IN Outgoing_Edge_List(node) LOOP
   IF (Node_Type(q->dest_node) == ASSERT_NODE &&
       (Edge_Type(q) == COND_EDGE || Edge_Type(q) == VALUE_EDGE)) THEN
    /* Case of node being a rhs source in an assertion */
    edge_tok := Symbolic_Matcher( q, Outgoing_Edge_List(q->dest_node) );
    dest_node := Get_Assertion_Target(q->dest_node);
   ELSIF (Edge_Type(q) == PARAM_EDGE &&
          Node_Type(q->dest_node) == LAST_NODE) THEN
    /* Case of edge into LAST */
    edge_tok := Get_Last_Relationship( q );
    dest_node := q->dest_node;
   ELSE
    /* DFA not interested in other edge types */
    CONTINUE;
   END IF
   IF ((p := Find_Edge_With_Target(cache, dest_node)) == NULL) THEN
    /* Add edge to cache */
    Push( Create_New_DFA_Edge(dest_node, edge_tok, q), cache );
   ELSE
    /* Combine multiple direct edges emanating from this node */
    Combine_Mult_Direct(p->dfa_tok, edge_tok);
   END IF
  END LOOP
  Set_Combined_Edge_List_Cache(node, cache);
 END IF
 RETURN cache;
END Combined_Edge_List;
```

reasoning for this ordering is fairly straightforward. Consider the case where at least one dimension of a direct edge has a dependency of the type I+1. This requires that all values on that dimension of the source node be computed before the current value of the target node on that dimension can be computed, hence it has the highest rank. Next consider the case where at least one dimension of a direct edge has a dependency of the type I with none of the type I+1. This requires that previous values as well as the current value on that dimension must be computed before the current value of the target node on that dimension can be computed. Hence, dimensions of type I have the second highest rank and dimensions of type I-1 the lowest rank. This same criterion is used when multiple paths exist between unique input and output port file fields.

The substitution of a single combined edge for multiple direct edges between two nodes need only be performed once since the information used creating this single edge is static. The function Combined_Depended_On_By_Edge_List is used to compute the combined edge list only once and caches the result for subsequent references.

The type of a single multi-edge path between two nodes in the array graph is the maximum dependency of each dimension (being considered) over all the edges in the path. The following order is used: I+1 (subscript types 0,4,8,9), I-1 (subscript types 2,3,6,7), I (subscript type 1,5). The reason for this ordering is slightly more complex than that for multiple direct edges. It is possible for an individual edge to have a sub-

set of the dimensions inherited by the input field from its parent record. When there is no dimension on a path's edge corresponding to a dimension of input field's parent record, this results in the type of this dimension of the path being I+1. This is because all values on this dimension are assumed to be needed to compute the current value of the target node of the subpath where this occurred.

Given that the edges in the new array graph will reflect the correspondence between dimensions of source and target in assertions, an explanation of the order may now be given. The explanation will take the form of an inductive proof for one dimension of the parent input record (the same holds true for each of these dimensions). It is obvious that a path of length 1 from an input field to another node in the array graph satisfies the criteria. Let's assume that this is also true of all paths of length k. We will show that the inductive hypothesis is true for a path of length k+1, thus proving it true for all paths of length k >= 1. For the proof we need to define the following symbols and terms (also see Figure 18).

D refers to the edge dimension between (k) and (k+1) nodes in the path we are examining.

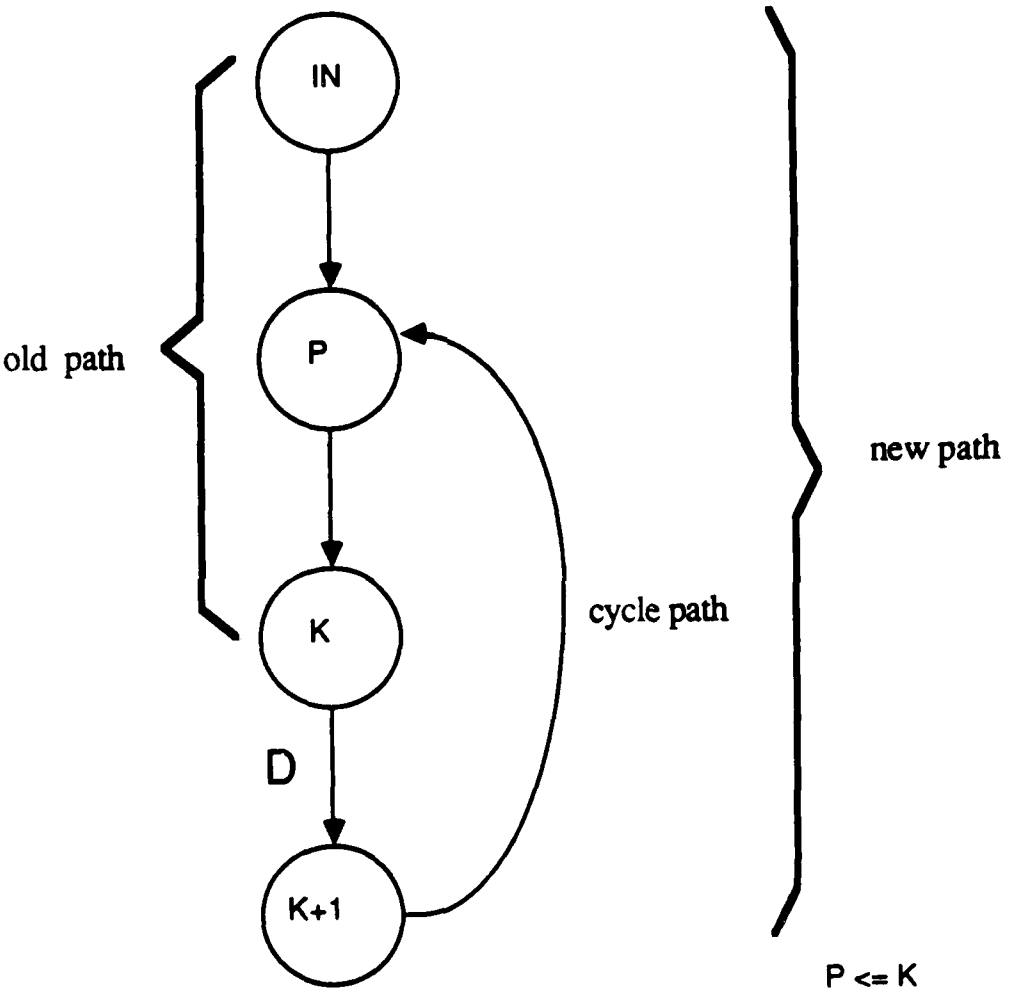
old path refers to the subpath (1, ..., k).

new path refers to the subpath (1, ..., k+1)

cycle path refers to the subpath (p, ..., k+1, ..., p) where p <= k.

[Note: The functions Type_Thru_Edge and New_Path_Type are used to perform the path typing whose description follows.]

Fig. 18 A graphical view of path typing



Case 1:

Consider the case where D is of type I+1. This requires that all values on that dimension be computed before the current value of the (k+1) node can be computed with

respect to D. Thus, the type of *new path* must become I+1 regardless of the type of *old path.* If a *cycle path* exists, via some p <= k, then all paths thru (p) become of type I+1 as well. Although this is implied from above, it is stated directly to emphasize that a cycle may affect a previously typed subpath.

Case 2:

Next consider the case where D is of type I-1. If *old path* is of type I+1 then so is *new path* (as explained in Case 1). The more interesting situation is when *old path* is of type I or I-1. This requires that previous values on D must be computed before the current value of node (k+1) can be computed with respect to D. Thus, when *old path* is not of type I+1, the type of *new path* must become type I-1. However, if a *cycle path* exists then there are 3 possible scenarios:

Case 2a:

When the type of the *cycle path* is I+1 we have the same situation as in Case 1 when a *cycle path* of type I+1 is present.

Case 2b:

If *new path* is of type I+1 then again we would refer back to Case 1.

Case 2c:

If *new path* is not of type I+1 and *cycle path* is of types I or I-1 then no changes need to be made. When the type *cycle path* is I-1 this is obvious. If the type of *cycle path* is I then we are in effect saying that the (p) node requires that the current value of the (k+1) node be computed and that the (k) node requires that some previous values of the (p) node be computed as well. This condition will be satisfied naturally when the required number of initial values of the (k) node are given (see Figure 19).

Fig. 19 Cycles in the Array Graph (example 1)

```
/* equation 1 */  A[I] = IN[I] + B[I];

/* equation 2 */  B[I] = A[I-1];
```

Case 3:

Finally consider the case where D is of type I. Given cases 1 and 2, the type of *new path* is obviously that of *old path*. However, if a *cycle path* exists then there are 3 possible scenarios:

Case 3a

When the type of the *cycle path* is I+1 we have the same situation as in Case 1 when a *cycle path* of type I+1 is present.

Case 3b:

When the type of the *cycle path* is I or I-1 and the type of *new path* is I+1 then all paths thru node (p) must become type I+1.

Case 3c:

In all other cases no changes are necessary. This is obvious when *new path* is of type I-1 and *cycle path* is of type I. When *new path* and *cycle path* are both of type I then there is a mutual value-for-value dependence between nodes (p) and (k+1) thus requiring the EPL compiler to generate a solution in the form of solving simultaneous equations (see Figure 20). However, these simultaneous equations can be solved for each input record. Thus, there is no need for any input record except the current one to be present in order to proceed.

It is important to note that the above description of deriving types through the symbolic matching of subscript references in an assertion is valid only after normalization of subscript references on the lhs of the equation [1]. The following equations are equivalent in saying that the 'next' occurrence is dependent on the value from a previ-

---

Fig. 20 Cycles in the Array Graph (example 2)

/* equation 1 */ A[I] = IN[I] + B[I];

/* equation 2 */ B[I] = A[I];

---

ous occurrence (they reference the array in ascending subscript order, ie. 1,2,3,...).

Case 1 (before normalization):

```
A[i] = A[i-k];
A[i+k] = A[i];
A[i+k] = A[i+k1];      where k,k1 > 0 and k > k1
A[i-k] = A[i-k1];      where k,k1 > 0 and k < k1
```

The following equations are equivalent in saying that the 'previous' occurrence is dependent on the next occurrence (the array must be calculated in descending subscript order, ie; 10,9,8, etc.).

Case 2 (before normalization):

```
A[i-k] = A[i];
A[i] = A[i+k];
A[i - k] = A[i - k1];   where k,k1 > 0 and k > k1
A[i + k] = A[i + k1];   where k,k1 > 0 and k < k1
```

From cases 1 and 2 it becomes clear that our rules for typing aren't consistent before normalization. It is not enough to look at an edge and simply test for i - k or i

+ k, the subscript type must be analyzed with respect to what is happening to it on the other side of the assertion. It must be found whether or not the lhs subscript expression is greater than or equal to the rhs subscript expression in order to determine the order in which the array is defined.

After normalization we would have the following:

Case 1 (after normalization):

```
A[i] = A[i-k]
A[i] = A[i-k]
A[i] = A[i-k2]        where k2 = k-k1 and k2 > 0
A[i] = A[i-k2]        where k2 = k1-k and k2 > 0
```

Case 2 (after normalization):

```
A[i] = A[i+k]
A[i] = A[i+k]
A[i] = A[i+k2]        where k2 = k - k1, k2 > 0
A[i] = A[i+k2]        where k2 = k1 - k, k2 > 0
```

At this point testing whether the lhs subscript expression > the rhs subscript expression or if the rhs subscript expression > the lhs subscript expression is easy. Edges of type i - k imply that you are processing the array in ascending subscript order, and edges of type i+k imply the opposite.

Complex subscript expressions will try to be normalized to one of the simpler expressions from above. After normalization the presence of a complex expression in a

subscript then would imply that the complete array must be present. If a complex subscript expression occurs in a recursive assertion (ie: A[i] = f(A[c1]), where c1 is a complex expression), then this would be considered to be an unbreakable cycle.

The output of the IDDBIOPS algorithm is directed to a file, to be used by the EDDA, whose name is the concatenation of the process name and the extension *.idd*. The format of the file is a list of all the output port files followed by a list of all the input port files and their dependent output port files. Note that since both port and direct file types may have only a single record format it is enough to store the file name and not the file name/record pair. The results of the IDDBIOPS analyses for the process specifications in the Dining Philosophers example are shown in Figure 21.

Fig. 21 IDDBIOPS Analyses for Dining Philosophers Example

```
/* p.idd */
O:REQ_REL 1     /* has 1 dimension */
I:ALLOC 1       /* has 1 dimension, 1 dependent output port file */
O:REQ_REL 0 1   /* dimension dependencies: type I-1 on 1st */

/* r.idd */
O:ALLOC 2       /* has 2 dimensions */
I:REQ_REL 1     /* has 1 dimension, 1 dependent output port file */
O:ALLOC 0 2     /* dimension dependencies: type I on 1st */
```

There are six possible relationships between an input port and output port of a process which the EDDA must recognize and handle. These relationships are expressed in terms of internal and external data dependencies between the input and output ports. In the following enumeration of the cases the input port is called MSGIN and the output port is called MSGOUT.

Case 1: No internal data dependency between MSGIN and MSGOUT.

(A) External data dependency of MSGIN on MSGOUT
An additional edge is added to the processes' array graph from MSGIN to MSGOUT

(B) NO external data dependency of MSGIN on MSGOUT
Array graph needs no patching.

Case 2: Internal data dependency of MSGOUT on MSGIN

(A) External data dependency of MSGIN on MSGOUT
Error possible, depending on the types of internal and external dependencies. If any of those dependencies is I+1 or none is I-1, then this configuration will result in a deadlock.

(B) NO external data dependency of MSGIN on MSGOUT
Just checking that configuration is error free.

Case 3: Internal data dependency of MSGIN on MSGOUT

(A) External data dependency of MSGIN on MSGOUT
Edge already in array graph because of internal data dependency, but the external dependency may introduce a higher rank for the type of the edge. Thus, the patch must be added to the array graph in this case.

(B) NO external data dependency of MSGIN on MSGOUT
Array graph needs no patching.

In cases *1A, 2A, and 3A* a significant configuration dependent result has been

achieved. New configurations need only be reanalyzed by the configurator and EDDA thus allowing fast prototyping of configurations. As stated earlier in the paper, in previous versions of EPL changing the configuration specification required the user to reanalyze all the process specifications. It involved adjusting the specifications by means of the DEPENDS_ON pseudo function, reinvoking the compiler on process specifications and then reinvoking the configurator.

The goal of the algorithm is to identify both external data dependencies between process' input and output port files as well as cycles in the super graph relative to the process' output port files. The procedure Get_Process_EDDS (see Figure 22) will determine the external data dependencies for a given configuration and process. As in performing the IDDBIOPS analysis for an array graph, the critical part of the external data dependency analysis is tracing paths through the super graph. This tracing is performed by the function Trace_Thru_Super_Graph which uses an algorithm identical to that of the function Trace_Thru_Array_Graph.

The type of a path in the super graph is determined in exactly the same way as used for typing paths in the array graph (see description of functions Type_Thru_Edge and New_Path_Type). Multiple external data dependencies between input and output ports are combined in the same manner as that described for combining multiple paths between input and output ports in the array graph (see description of function

Fig. 22 Procedure Get_Process_EDDS

```
PROCEDURE Get_Process_EDDS( configuration, process )
  FOR a_process IN Configuration_Process_List( configuration ) LOOP
    Create_Node_In_Super_Graph( a_process )
  END LOOP
  FOR edge IN Configuration_Edge_List( configuration) LOOP
    Add_Edge_To_Super_Graph( edge )
  END LOOP
  FOR out_port IN Output_Ports( process ) LOOP
    UnMark_Super_Graph
    dependency_list := NULL
    Trace_Thru_Super_Graph( out_port )
    FOR dependency IN dependency_list LOOP
      Output dependency
    END LOOP
  END LOOP
End Get_Process_EDDS;
```

Combined_Edge_List). However, if a cycle is found in the super graph which is not of type I-1 then it is reported as an error (ie. the given configuration and process specifications would result in an unschedulable global computation).

One important difference, however, between the array graph and super graph is that all edges in the array graph have explicit types while some edges in the super graph have implied types. These are the edges which are taken from the configuration specification. Since messages sent via port files are queued at the receiver, edges in

the configuration specification are always assigned type I.

It is important to notice that the EDDA performs an important semantic check on the configuration specification. It determines if the message formats used by two communicating processes are compatible in terms of their dimensionalities. This is in addition to its primary goal of deriving external data dependencies.

As previously stated the EDDA generates a *patch* file for use by the intra-process equation scheduler in the Comp2 phase of the compiler. This file's name is the concatenation of the configuration name, the process's name and the extension *.edd*. For example, if the configuration's name is CONFIG_1 and the process' name is P1 then the file would be named, config_1.p1.edd. Since the same process may be part of several different configurations, then several different patch files may exist for the same specification. The format of this file is a list of the additional internal data dependencies (edges and their dimension types) that need to be added to the process' array graph. The patch files generated by the EDDA for the processes in the Dining Philosophers example are shown in Figure 23.

The Trace_Thru_Super_Graph algorithm additionally provides several checks to ensure that all processes named in the configuration have IDDBIOPS, and that they have not been tampered with.

Fig. 23 EDDA Patch Files for the Dining Philosophers Example

```
/* patch file for P */
X:ALLOC REQ_REL 0 2          /* ALLOC depends on REQ_REL by I */

/* patch file for R */
X:REQ_REL ALLOC 0 1          /* REQ_REL depends on ALLOC by I-1 */
```

Limitations

Presently the algorithm to propagate external data dependencies has two limitations. The first is crude analysis of I+1 type dependencies. The algorithm assumes that an I+1 type dependency on a path between input and output ports implies reading in all messages before writing out any messages. This may prevent the best possible schedule from being generated or cause no schedule to be generated. The author realizes that this is a problem and is currently investigating a more complete solution where the specific values of constants in relations of subscript types 3,4,5,7,8 will be considered.

The other limitation of this algorithm is that it regards all dependencies as unconditional. The array graph does not record conditions under which dependencies hold. Therefore, when conditional data dependencies are traced through a process' array graph to determine its IDDBIOPS, the algorithm will also treat them as always valid.

This may prevent the best possible schedule from being generated or cause the detection of the possibility of a deadlock which, if the fact that the paths were conditional was considered, would be found impossible.

The Implementation

The EDDA has been implemented in C under UNIX. However, as discussed in the section "The New EPL System," there is a need for a facility to save/restore the array graph and its supporting structures (symbol tables, equation trees) to/from disk. With such facility the structures created by the Comp1 stage can be restored by the Comp2 stage saving repeating processing of the same specification by Comp1. This facility was missing from EPL and neither the C language nor UNIX provide a mechanism to save/restore arbitrary data structures created by a program to/from disk.

Since this is a reoccurring problem when programming in C under UNIX, the need for a tool which could automatically add this capability to an existing C program became immediately apparent. As a part of the reported EPL enhancement the author developed the Data Structure Analyzer (DSA).

The DSA takes as input the C source file(s) that comprise a single program and analyzes its data type and storage declarations. After completing a successful analysis, the DSA will generate a C module which, when linked with other modules which share these same definitions, allow that system to dump/restore any of its data

structures to/from disk. The DSA does have some limitations. However, these limitations are not exceeded by the EPL compiler or typical C programs, so the DSA can be safely used in the EPL system.

The DSA performs its analysis on data type and storage declarations. There are four ways a new data type may be declared in C: ENUM, UNION, STRUCT and TYPEDEF. The DSA compiles all type declarations into a digraph which represents each type's hierarchical structure and its dependency on other types. If there are no references to undeclared types, this graph is used to generate the dump/restore code. The graph completely captures all static semantics of the system's type definitions. The generated code must be able to access dynamic attributes for handling unions and arrays without fixed dimensions.

To this end the DSA provides a substitute interface to the UNIX allocation and deallocation routines. This interface maintains dynamic tables which, when used in conjunction with the type hierarchy and dependency graph, provide the generated dump/restore subsystem with all the necessary information to perform its task.

The DSA requires the user to make only minor modifications to the source program. First, all references to the standard allocation and deallocation routines must be changed appropriately. Second, the header file which is generated by the DSA must be included in any source file which performs allocation/deallocation or the

dumping/restoring of data structures. Finally, the user must elaborate how the DSA can dynamically determine which component of a *union* is in use. For this purpose the C language syntax has been extended to allow for the following union declarator:

union { ... } UTAG(expression) union_declarator

where expression will evaluate to the position of the union component (starting from 0) which is presently in use. The user may reference the pseudo-variable $ to refer to the type definition in which this union is contained (see example in Figure 24).

Fig. 24 A Sample C type definition using the extended union syntax

```
struct x {

  short type;                  /* 1 if char, 2 if int, 3 if float */
  union {
    char  a[10];
    int   b[10];
    float c[10];
  } UTAG(($.type - 1)) p;      /* here $ refers to the instance of
                                  struct x containing the union */
} example;
```

The system provided header file *dsa.h* has to be included in any source program which uses the extended syntax (actually, *dsa.h* may be safely included in every source program). Once the file containing the dump/restore routines has been

generated it is simply compiled along with the other files and the DSA run-time library. The run-time library handles the machine-level dependencies the DSA needs to be aware of when dumping/restoring structures to/from disk.

Appendix I: Philosopher Process Specification in EPL

```
PROCESS: P[*];
IN: ALLOC;
OUT: REQ_REL;

FILE: ALLOC (PORT),
  10 RECORD: MSGA[*],
    20 INT: PROC_ID,
    20 INT: CLOCKA;

FILE: REQ_REL (PORT),
  10 RECORD: MSGR[*],
    20 INT: PROC_ID,
    20 LOGIC: RQ_OR_RL,
    20 INT: RES[5],
    20 INT: CLOCKR;

SUBS: I, J;
SUBS: IX SUBLINEAR (I) ~RQ_OR_RL[I];

RANGE.MSGR = IF (RQ_OR_RL[I] & RANDOM() > .99) THEN I ENDIF;

LASTALLOC[I] = CLOCKA[IX];
PID = PROCID;

REQ_REL.PROC_ID[I] = PID;
RQ_OR_RL[I] = ~RQ_OR_RL[I-1];
RES[I,J] =  IF ((J==PID) | (J==PID+1) | (J==PID-4)) THEN 1 ELSE 0 ENDIF;
CLOCKR[I] = IF (I==1) THEN
              TIME()
            ELSIF (RQ_OR_RL[I]) THEN
              LASTALLOC[I-1] - 1000 * LOG(RANDOM())
            ELSE
              CLOCKR[I-1] - 10000 * LOG(RANDOM())
            ENDIF;
```

Appendix II: Resource Allocator Process Specification in EPL

```
PROCESS: R;
IN: REQ_REL;
OUT: ALLOC, QUEUE;

FILE: REQ_REL(PORT),
  10 RECORD: MSGR[*],
     20 INT: PROC_ID,
     20 LOGIC: RQ_OR_RL,
     20 INT: RES[5],
     20 INT: CLOCKR;

FILE: ALLOC (PORT),
   10 RECORD: MSGA[*,*],
      20 INT: PROC_ID,
      20 INT: CLOCKA;

FILE: QUEUE (SEQ),
   10 RECORD: PROCC[*,*],
      20 INT: PROC_ID,
      20 INT: IN_IX,
      20 INT: OUT_IX,
      20 GROUP: RES[5],
         30 INT: CLAIM,
         30 INT: SUM_CLAIM,
         30 LOGIC: SAT;

INT: NUM_RES[5];

SUBS: I, J, K, L;

RANGE(2).MSGA = 100;                    /* must declare # of msgs */

RANGE.PROCC[I] = IF (I==1) THEN
                    1
                 ELSIF (RQ_OR_RL[I]) THEN
                  RANGE.PROCC[I-1]-1
```

```
            ELSE
             RANGE.PROCC[I-1]+1
            ENDIF;

RANGE.MSGA[I] = IF (RANGE.PROCC[I] > 0) THEN LAST.OUT_IX[I] ELSE 0 ENDIF;

NUM_RES[J] = 1;

QUEUE.PROC_ID[I,K] = IF (~RQ_OR_RL[I] & (K==RANGE.PROCC[I])) THEN
                       REQ_REL.PROC_ID[I]
                     ELSE
                      QUEUE.PROC_ID[I-1, IN_IX[I, K]]
                     ENDIF;

IN_IX[I,K] = IF (~RQ_OR_RL[I]) THEN
              K
             ELSIF (REQ_REL.PROC_ID[I] ~= QUEUE.PROC_ID[I-1, K]) THEN
              IN_IX[I,K-1]+1
             ELSE
              IN_IX[I,K-1]+2
             ENDIF;

OUT_IX[I,K] = IF RQ_OR_RL[I] THEN
               IF (LAST.SAT[I, K] & ~LAST.SAT[I-1, IN_IX[I,K]]) THEN
                 OUT_IX[I, K-1]+1
               ELSE
                 OUT_IX[I, K-1]
               ENDIF
              ELSE
               IF (K==RANGE.PROCC[I]&LAST.SAT[I, K]) THEN 1 ELSE 0 ENDIF
              ENDIF;

CLAIM[I, K, J] = IF (~RQ_OR_RL[I] & (K==RANGE.PROCC[I])) THEN
                   REQ_REL.RES[I, J]
                 ELSE
                   CLAIM[I-1, IN_IX[I, K], J]
                 ENDIF;

SUM_CLAIM[I, K, J] = QUEUE.CLAIM[I, K, J]+SUM_CLAIM[I, K-1, J];
```

```
SAT[I, K, J] = SAT[I, K, J-1] &
            (SUM_CLAIM[I,K,J] <= NUM_RES[J] | QUEUE.CLAIM[I, K, J]==0);

ALLOC.PROC_ID[I, OUT_IX[I, K]] = IF (OUT_IX[I,K] > OUT_IX[I,K-1]) THEN
                                 QUEUE.PROC_ID[I, K]
                               ENDIF;

CLOCKA[I, L] = CLOCKR[I];
```

References Cited

[1] Bruno, J. M., "Notes from EPL group meeting on August 24, 1988," Department of Computer Science, Rensselaer Polytechnic Institute, August 1988.

[2] Clarke, D. E., "EPL System Design -- Intermediate Processing Steps," Masters Project, Department of Computer Science, Rensselaer Polytechnic Institute, December 1987.

[3] Hoffman, C S. and O'Donnell, M. J., "Programming with Equations," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, pp. 83-112, January 1982.

[4] McGraw, J. R., "The VAL Language: Description and Analysis," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, pp. 44-82, January 1982.

[5] Ramamrithan, K. and Keller, R. M., "Specification of Synchronizing Process," IEEE Transactions on Software Engineering, Vol SE-9, No. 6, pp. 722-733, November 1983.

[6] Shi, Y., Prywes, N., Szymanski, B., and Pnueli, A., "Very High Level Concurrent Programming," IEEE Transactions on Software Engineering, Vol. SE-13, No. 9, pp. 1038-1046, September 1987.

[7] Srinivasan, Mahesh, "A Timing Evaluator for C Programs Generated by the MODEL System," Technical Report submitted to Information System Program, Office of Naval Research, under contract N00014-86-K-0442.

[8] Szymanski, B., "Parallel Programming with Recurrent Equations," International Journal of Supercomputer Applications, Vol. 1, No. 2, pp. 44-74.

[9] Tseng, J.S., Szymanski, B., Shi Y., Prywes, N., "Real-Time Software Life Cycle with the Model System," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, pp. 358-373, February 1986.